Computation and Visualisation in the NumLab Numerical Laboratory

Maubach, J.M.L.; Telea, A.C.

Published in:
EPRINTS-BOOK-TITLE

Computation and Visualisation in the NumLab Numerical Laboratory

J.M.L. Maubach[1] and A.C. Telea[1]

Eindhoven University of Technology, Department of Mathematics and Computer Science, Postbox 513, NL-5600 MB Eindhoven, The Netherlands
j.m.l.maubach@tue.nl, a.c.telea@tue.nl

Summary. A large range of software environments addresses numerical simulation, interactive visualisation and computational steering. Most such environments are designed to cover a limited application domain, such as Finite Elements, Finite Differences, or image processing. Their software structure rarely provides a simple and extendible mathematical model for the underlying mathematics. Assembling numerical simulations from computational and visualisation blocks, as well as building such blocks is a difficult task.

The NumLab environment, a numerical laboratory for computational and visualisation applications, offers a basic, yet generic and efficient framework for a large class of computational applications, such as partial and ordinary differential equations, non-linear systems, matrix computations and image and signal processing. Building applications which combine interactive visualisation and computations is provided in an interactive visual manner.

This paper focuses on the efficient implementation of one of the most complex NumLab components, the Finite Element assembler for systems of equations, such as Stokes or Navier-Stokes fluid-flow equations. It shows how the software framework as a whole has been targeted towards fast assemblers, and how a general purpose fast Finite Element assembler is embedded.

1 Introduction

As pointed out in [1], the NumLab (Numerical Laboratory) environment has been constructed after a thorough search through a wide range of software environments for numerical computation and data visualisation. The NumLab design goals included a seamless integration of computation and visualisation, as well as convenient computer aided application construction. Furthermore, all components should be customisable and nevertheless fast.

Paper [1] explains the design concepts in detail: The mathematics to be modeled (section 2), a software framework which is one to one with the mathematical model (section 3), and discusses the implementation of a Navier-Stokes Finite Element solver within the NumLab software framework (section 4).

Over time, the large scope of NumLab's mathematical model – which covers numerical methods for ODEs, PDEs, non-linear systems, and all possible combinations – has induced a number of questions. Most questions have either been on mathematical framework (addressed in [1]), the technical realisation (the construction of libraries is addressed in [2]), and on overall efficient **and** customisable implementations, addressed in this paper.

This paper shows how NumLab's most complex component (the Finite Element assembler for non-linear systems) can be implemented such that it is both fast and customisable. On the small-detail level, introducing general purpose techniques, we demonstrate how NumLab's Finite Element assembler can be efficient (fast). The introduced techniques can also be used for Finite Difference and Volume assemblers. Next, we show the construction of an efficient *and* customisable Finite Element assembler for systems of equations.

The NumLab environment consists of two parts. The first part is its content, i.e., its c++ libraries. Standard c++ programs can be written using NumLab's libraries (data and operations) and can be compiled as well as interpreted (interpretation using the CINT c++ interpreter [5]). The second part of NumLab is its interactive program creation and simulation tool called vission (see [11, 12]). With this tool, standard c++ programs can be composed using NumLab's libraries (data and operations) in a data-flow graphical environment. This environment is well-suited for numerical and visual simulations.

There are two categories of NumLab libraries: *base* and *derived* libraries.

The base NumLab libraries are unaltered public domain and commercial (fortran, pascal, c, c++) libraries (binaries) *adorned with*:

1. a small c++ interface for the communication of standard data types between the different libraries;
2. a smaller c++ interface for computer aided program creation and simulation (see [2]);
3. an added so-called reflection layer (see [5]) for the c++ interpreter, and;
4. a smallest data-flow simulation interface.

There are libraries for computation – LAPACK [4], NAGLIB [20], or IMSL [16], SEPRAN [17] – as well as for professional visualisation – OpenGL [8], Open Inventor [14], or VTK [9]. The (parts of) libraries which are available is research-determined, for instance OpenDX [15] is not available in NumLab. The NumLab base libraries communicate with the use of standard data types, which have been quite stable for a period of years: Open Inventor and VTK types are used for visualisation, c++-wrapped BLAS types (Basic Linear Algebra Subprograms, part of LAPACK) are used for computation with full matrices, and in-house data types are used for sparse matrix operations.

Also Boost data types (`http://www.boost.org/`) are used. At this moment, O(1000) data types and operators are available. The data types have readers and writers for VTK, LAPACK, Matlab, and sometimes LaTeX, OpenMath, and MathML formats.

The derived NumLab libraries implement fundamental mathematical notions such as `operator` and derived `solver`, `operator_pde` and `operator_ode`, as well as the data to operate at: linear vector spaces of functions `space` with elements `Function`. Similar mathematical concepts are factored out into similar orthogonal software components, so few components can be combined to powerful solution algorithms. An example is shown in section 5. The derived NumLab libraries communicate with the data types `Function` and `Operator`.

In one aspect, the NumLab graphical editor `vission` resembles other graphical editors such as Matlab's Simulink [18], AVS [13], IRIS Explorer [3] or Oorange [7]. In each case, data and operations from libraries are represented as rectangles to be put on a canvas, and input and output arguments can be connected.

However, in contrast with the other graphical editors, VISSION steers a c++ interpreter. Thus all kinds of c++ expressions *could be* – and in a few cases are – added in adornment-level (4), in order to implement powerful features at minimal cost. Furthermore, the use of a single language for content (libraries) and management (graphical editor) simplifies the implementation of the NumLab workbench.

For a more detailed information and a comparison of NumLab with Matlab [18], Mathematica [19], Diffpack and SciLab [6, 21], see [1].

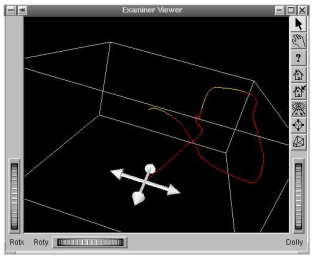
Uniting visualisation and numerical frameworks turns out to be powerful: `element` and derivatives of `operators` are visualised without effort because (1) elements support sampling and (2) derivatives of operators are represented using matrices (see section 3). All Open Inventor and VTK functions (contour surfaces, probing, Fast Fourier transforms, etc.) are available for elements (which are vector-valued functions), and even mpeg generation modules are available in NumLab. Table 1 shows a few examples:

1. A matrix (an image): values determine the color;
2. Interactive computation of streamlines using Open Inventor probes;
3. Axes with names and labels in 2 and 3 dimensions;
4. Computer aided design;
5. Feature detection in complex 2 and 3 dimensional vector (flow) fields;
6. Computer aided design (extrusion) post-processes numerical output;
7. A matrix (linear operator): values determine height and color;
8. A matrix: sparsity pattern;
9. Automatic generation of editors to edit network default values;
10. A VTK-generated tetrahedral grid;
11. Contour surfaces;

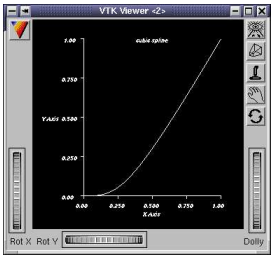
12. Streamlines.

Table 1. NumLab Visualisation of various computational results

1

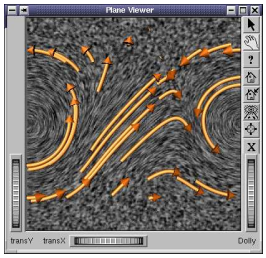


2

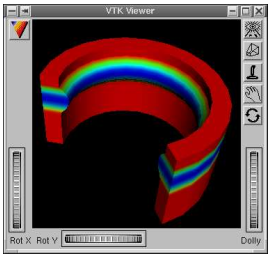


3

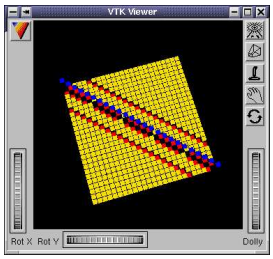
4



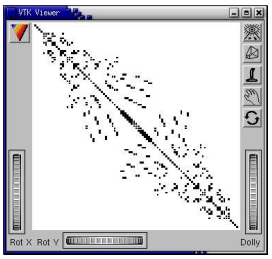
5



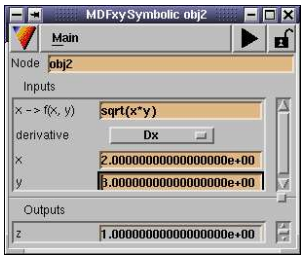
6



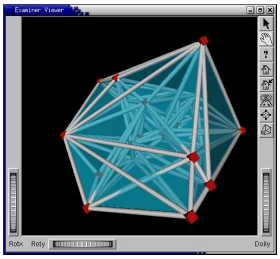
7



8

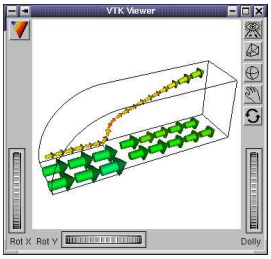


9



10

11



12

The remainder of this paper is organised as follows. Section 2 mentions the mathematics which can be modeled with the NumLab software framework. Next, section 3 formulates the NumLab software framework. Each component and its mathematical concept is described. Further, section 4 comments on the efficient implementation of a Finite Element Solver for systems of equations such as the Navier-Stokes equations. Then section 5 shows how to construct simulation applications from the modules introduced in section 3, using the graphical editor. Section 6 provides conclusions and discusses future developments.

2 The Mathematical Framework

The NumLab software framework models the mathematical notions *element* of a linear vector space, and *operator* on an element of such a space. In particular, elements are linear combinations (of functions), and are vector valued. The supported operations are vector space operations and function evaluations:

1. The addition of elements;
2. The scaling of elements;
3. The evaluation at elements $u = F(v)$ resulting in an element, and;
4. The evaluation of the derivative $G = \partial F(v)$, resulting in an operator.

The fundamental ideas behind the NumLab framework are:

1. all important numerical problems can be formulated as: Find the solution v of $F(v) = 0$;
2. most numerical methods induce a sequence of states

$$v^{(0)} \mapsto v^{(1)} \mapsto v^{(2)} \mapsto v^{(3)} \mapsto \dots \tag{1}$$

Related to each transition $v^{(k)} \mapsto v^{(k+1)}$ is an operator which can be expressed using the four operations mentioned above. Even for complex applications (see the Finite Element solver for the Navier-Stokes equations in section 5), it is possible to formulate **all** transitions from $v^{(0)}$ down to the final solution approximation $v^{(m)}$ with the use of just one transition operator F. Thus, in NumLab, implementations for complex problems can be composed in a simple manner.

In [1], it is shown how all of the following mathematical entities can be formulated as operators:

1. Transient boundary value problems (IBVPs);
2. Boundary value problems (BVPs)/Initial value problems (IVPs);
3. Systems of (non-)linear equations;

4. Iterative solution methods;
5. Preconditioners, etc.

The simple but powerful concept of `operator` makes NumLab a versatile workbench which is simple to use. But, it also has generated a lot of questions such as (1) how to map for instance a Finite Element assembler onto this framework and (2) how to do this in a run-time efficient **and** customisable manner. Paper [1] answers questions with respect to (1) for concrete examples of the above operators, and this paper addresses (2) for the most complex NumLab operator: A Finite Element BVP solver for systems of non-linear equations. An efficient implementation turns out to demand a lot of attention to small detail as well as to larger scale software design. Both aspects are addressed in section 4, using the modules described in section 3.

3 The Software Framework

This section describes the software framework which models two major notions: Element `Function x` of a set, and operator `Operator F` on such sets. These notions are sufficient for the solution of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$. The names of the methods and variable passing techniques are data flow framework standards (see [1] [2]). Below, the modules introduced in [1] are presented in a form which is more suitable for the explanation of an *efficient* Finite Element implementation in section 4. To this end, module `Space` is split into `Space` and `Basis`.

3.1 The `Function` module

An instance v of `Function` is vector of functions from $\Omega \subset \mathbf{R}^n$ to $\mathbf{R}$:

$$v_i(\mathbf{x}) = \sum_{j=1}^{N_i} v_{ij} \phi_j^{(i)}, \tag{2}$$

where all $\phi_j^{(i)} : \Omega \mapsto \mathbf{R}$. Set $N := \sum_i N_i$. Thus, `Function` contains (a reference to) a vector of coefficient vectors $\mathbf{v}_i = [v_{i1}, \ldots, v_{iN_i}]$, as well as (a reference to) a set of functions, called `Space`. The `Function` module provides a few services: evaluation (*sampling*) of v, as well as of its first and second derivatives at (collections of) points of $\mathbf{R}^n$. In fact, `Function` contains a sequence of elements related coefficient vectors $\{\mathbf{v}^{(k)}\}_k$, either because the user or an operator (for instance a time-integrator) issues a request to this end. In retrospect, from the authors point of view, `Function` had better been called *LinearCombinationOfFunctions*, *Element* or just *Data*. Nevertheless, we use `NumLab`'s name `Function` through the remainder of this paper.

3.2 The `Space` module

The evaluation of a `Function` instance v at point $\mathbf{x}$ is delegated to the `Space` module because the latter "contains" all functions $\phi_j^{(i)} : \Omega \mapsto \mathbf{R}$. For an efficient implementation, $\mathbf{v}$ and $\mathbf{x}$ are passed to `Space`. The module `Space` contains a sequence of `Basis`. A `Basis` can implement pre-run-time determined functions such as $\phi_j^{(i)}(\mathbf{x}) = |\mathbf{x}|^2$, but can also contain run-time determined functions, which is the case for Finite Element implementations. Because Finite Element bases require computational grids, a `Space module` contains (a reference to) a `Grid` module. Different bases (constant, linear, etc., conforming/non-conforming), can be chosen and altered during runtime. `Space` delegates the evaluation services of `Function` to its `Basis` (plural), which is given `Grid`. In this manner, `Basis` can call on `Grid` to determine the location of the point $\mathbf{x}$ and determine the related value $v(\mathbf{x})$.

The most important service of `Space` is access to the assembler, with Finite Element computations in mind. Passed an `Equation` from an `Operator-ImplementationFiniteElement` (as in figure 1, box 3), `Space` delegates the assembler request – in a loop over all solution components – to each `Basis`, passing the other involved `Basis`, the `Grid`, the `Equation`, as well as the `Function` v where F or its derivative must be evaluated. For more detailed information, see 4.

The boundary conditions `BoundaryConditions` are members of module `Space`, though in retrospect a more logical place would have been module `Contour`, which describes the boundary $\partial\Omega$, used by module `Grid` for the grid generation. In retrospect, a more mathematical name for `Space` would have been *CollectionOfFunctions*.

The services of `Basis` can be provided through `Space` (see [1], where `Basis` is not mentioned). However, in order to demonstrate that an efficient but flexible implementation is possible, `Basis` is regarded as a separate module.

3.3 The `Basis` module

Specialisations of this module implement a Finite Element basis $\{\phi_j^{(i)}\}_{j=1}^{N_i}$. The NumLab workbench offers the most common bases: Conforming piecewise constant, linear, quadratic, cubic, bilinear, biquadratic (etc.), and nonconforming piece-wise linear. In order to ensure that `Basis` can offer its services, `Space` invokes a method `Basis::init(Grid)`, where `Basis` generates all information it needs (amount of basis functions, support points, etc.).

There are two services: (1) Returning function values when passed a coefficient vector $\mathbf{v} = [v_1, \ldots, v_{N_i}]$ and collection of points $\mathbf{x} \in \Omega$; (2) Returning $F(v)$ or $\partial F(v)$ when passed an `Equation` (see module `Operator` below). In fact, `Basis` delegates the Finite Element assembler service in a non-trivial manner, as described in detail in section 4.

3.4 The `Grid` module

In order to generate (representations of) functions $\phi_j^{(i)} : \Omega \mapsto \mathbf{R}$ for Finite Element and other numerical computations, one needs a so-called computational grid. The domain of interest Ω is partitioned into a collection of non-overlapping elements e_l (spectral Finite Element methods require just one element). In the NumLab framework, a `Grid` module can contain a computational grid for $\Omega \subset \mathbf{R}^n$, where n can be adapted run-time. Available are regular grid generators – n-cube, n-simplex, or n-prism – and Delaunay grid generators – triangle, tetrahedron (see figure 1 j). The few services of grid are: Location of the elements in which a collection of points is situated (not public, but important intern for interpolation issues), and ensuring the element topology and data is available to module `Basis`. As a matter of fact, `Grid` takes a `Contour` as input, which describes the boundary $\partial\Omega$ of the region Ω.

3.5 The `Operator` module

In NumLab, all problems such as IVPs and BVPs are expressed using `Operator` modules. Also (iterative) solvers and preconditioners are expressed using an `Operator` module. Each `Operator` F module provides two services: (1) Evaluation at element `Function` v; (2) Evaluation of the derivative of F at v. The first operation returns a `Function`, the latter an `Operator`. `Operator`s can delegate (use) to `Operator`s in order to solve complex problems.

As an example, consider the NumLab Finite Element operator F: `Operator-ImplementationFiniteElementGalerkin` It has (a reference to) a module `Equation`, which specifies the components of the involved partial differential equations, and (a reference to) a module `Function`, an optional predetermined solution of the BVP. Upon evaluation at element `Function` v, F passes its `Equation` member to v's `Space` member, which passes it to its `Basis` members. For further explanation, see section 4. The determination of the operator $F'(v)$ is done in a similar manner.

3.6 The `Solver` module

Iterative solvers, for instance for the solution of a system of equations $Ax = b$, are described with the use of operators which are of the form

$$(v^{(0)}, b, A) \mapsto v^{(k)}, \tag{3}$$

where $v^{(k)}$ is the first iterand such that $\left|Av^{(k)} - b\right| < \epsilon$. Here both $v^{(0)}$ and b are `Function`s and A is an `Operator`. The operator in (3) need not be differentiable. Therefore, a `Solver` module, which represents an iterative solver, could be derived from `Operator`, and return a zero derivative. In fact, `Solver` is derived from `OperatorIterator` which specialises `Operator` in

that is has a member of `IterationControl`, which stores: maximum amount of iterations, convergence tolerance, etc. When requested for a derivative, it returns the zero `Operator`. As shown in figure 1, solvers can delegate (use) to solvers in order to solve complex problems.

In retrospect, the best modular approach would have been module *Operator* with evaluation service, and a derived module *OperatorDifferentiable* with added evaluation of derivative service.

4 An Efficient NumLab Finite Element Implementation

An efficient Finite Element assembler requires vector, matrix and sometimes tensor operations. To this end, in NumLab, three groups of such storage classes are required:

1. "small" storage classes (vector, matrix, tensor) for:
   a) points in a region $\Omega \subset \mathbf{R}^D$;
   b) values of T reference basis functions on Q quadrature points in $\mathbf{R}^{T\times Q}$, etc.;
2. "large" storage classes (vector, matrix) for:
   a) coefficient vectors $\mathbf{v}$ of a component of a `Function`;
   b) Jacobian matrices components of the derivative of an `Operator` evaluated at a `Function`, etc.;
3. "block" storage classes for composed items:
   a) block vectors of large vectors for the sequence of coefficient vectors of a `Function`;
   b) block Jacobian matrices of the derivative of an `Operator` evaluated at a `Function`, etc.;

In NumLab, the assembler's numerical integration routines use the small storage class, the computed entries are stored in large storage class, through a block storage class interface. The different storage classes have different services. The small storage class has just a few services which include addition, scaling, increment and specialised versions of matrix inversion for 2 x 2 and 3 x 3 matrices. The large matrix class has additional services such as for instance ILU(0) (incomplete LU factorisation).

First, we consider a scalar equation, related to a convection diffusion problem. After a step for step procedure, we arrive at a first Finite Element implementation, and discuss the reasons for its slow performance. The problem of interest is:

$$L(u) = f \quad \text{in } \Omega, \qquad \text{and} \qquad B(u) = g \quad \text{at } \partial\Omega, \tag{4}$$

where

$$L(u) = -\nabla\cdot a\nabla u + \mathbf{b}\nabla u + cu, \qquad B(u) = u, \tag{5}$$

and with the domain of interest $\Omega \subset \mathbf{R}^D$. The diffusion coefficient a, convection vector $\mathbf{b} \in \mathbf{R}^D$ and source coefficient c are all functions of $\mathbf{x} \in \Omega$. For the sake of demonstration, we assume that the domain Ω is covered with a computational grid of elements e. Assume that, on the reference element, we use Q quadrature points $\mathbf{x}_k$ and related weights w_k. Abbreviate $a_k := a(\mathbf{x}_k)$, $\mathbf{b}_k := \mathbf{b}(\mathbf{x}_k)$ and $c_k = c(\mathbf{x}_k)$.

There are two major types of Finite Element assemblers:

1. Galerkin-type Finite Element assemblers, which must integrate:

$$\int_e a\nabla\phi_s\nabla\phi_r + \phi_r\mathbf{b}\nabla\phi_s + c\phi_s\phi_r. \tag{6}$$

   These integrals are approximated with the use of numerical integration:

$$\sum_k w_k \left[a_k\nabla\phi_s(\mathbf{x}_k)\nabla\phi_r(\mathbf{x}_k) + \phi_r(\mathbf{x}_k)\mathbf{b}_k\nabla\phi_s(\mathbf{x}_k) + c_k\phi_s(\mathbf{x}_k)\phi_r(\mathbf{x}_k)\right], \tag{7}$$

   for all $r, s = 1, \ldots, T$, the amount of basis functions ϕ_r on the reference element specified through `Basis`;
2. Petrov-Galerkin-type Finite Element assemblers, which must integrate:

$$\int_e a\nabla\psi_s\nabla\phi_r + \phi_r\mathbf{b}\nabla\psi_s + c\psi_s\phi_r. \tag{8}$$

   These integrals are approximated with the use of numerical integration:

$$\sum_k w_k \left[a_k\nabla\psi_s(\mathbf{x}_k)\nabla\phi_r(\mathbf{x}_k) + \phi_r(\mathbf{x}_k)\mathbf{b}_k\nabla\psi_s(\mathbf{x}_k) + c_k\psi_s(\mathbf{x}_k)\phi_r(\mathbf{x}_k)\right], \tag{9}$$

   for all $r = 1, \ldots, T_1$, and $s = 1, \ldots, T_2$ the amounts of basis functions ϕ_r and ψ_s on the reference element specified through `Basis_1` and `Basis_2`.

Systems of non-linear equations such as the Navier-Stokes equations can be solved using both:

1. The Galerkin-assembler, if all different solution component bases are merged into one `Basis`, and;
2. The Petrov-assembler for each block-component of $F(v)$ and $DF'(v)$.

The `NumLab` framework uses the latter approach because it deals better with the zero entries in the derivative (as is the case for Stokes and Navier-Stokes).

Now, consider a straightforward implementation of (7). Assume we precompute and store all values $\phi_r(\mathbf{x}_k) \in \mathbf{R}^{T\times Q}$ in matrix `v_e`, $\nabla\phi_r(\mathbf{x}_k) \in \mathbf{R}^{D\times T\times Q}$ in tensor `grad_v_e`, $a(\mathbf{x}_k) \in \mathbf{R}^Q$ in vector `a`, etc. Then, the common implementation of (7) is:

```
for (int k = 0; k < Q; k++)
 for (int r = 0; r < T; r++)
  for (int s = 0; s < T; s++)
   a_rs(r, s) += w_k(k) * (
    a(k) * (grad_v_e(s, k) * grad_v_e(r, k)) +
    v_e(r, k) * (b * grad_v_e(s, k)) +
    c * v_e(s, k) * v_e(r, k)
                          );
a_rs *= abs(detAe);
```

The problem: The implementation is observed to be slow, if vector, matrix and tensor classes use run-time allocated storage (which is the case if a constructor such as `smallvector::smallvector(n)` exists). In this case, calls to `new` are slow on at least some machines, and bounds must be checked for safe selections `a(k)`. The case of multiple selections (selection of `grad_v_e(s, k)` in block `(p, q)` of a Jacobian at element face number `f`) causes the program to almost stand still. The added problem is that the *implementer* is to choose a small data storage class because c and c++ have no built-in types for such containers. Worse, because the built-in support lacks, c and c++ compilers can not optimise for it – Fortran compilers can.

The first step towards an efficient implementation: Ensure that the assembler for one component (or block component is fast). Implement the "small" storage class with type-`int` templates. This is *the only* manner to avoid run-time allocated storage, but bound checks would still be needed when selections are to be performed in a safe manner. However, such checks are not required if iterators are provided for the "small" storage class, and used (actual NumLab code):

```
const double *wk = w_k.begin();
const double *ak = a.begin();
const smallVector<T> *v = v_e.begin();
const smallMatrix<D, T> *gradv = grad_v_e.begin();
// loop over quadrature points
for ( ; wk < w_k.end(); wk++, ak++, v++, gradv++)
{
 double *ars = a_rs.value();

 const double *vs = v->begin();
 const smallVector<D> *gradvs = gradv->begin();
 // loop over trial function s
 for ( ; gradvs < gradv->end(); vs++, gradvs++)
 {
  const double *vr = v->begin();
  const smallVector<D> *gradvr = gradv->begin();
```

```
 // loop over test function r
 for ( ; gradvr < gradv->end(); vr++, gradvr++)
  *ars++ += (*wk) * (
                     (*ak) * ((*gradvr) * (*gradvs)) +
                     (*vr) * (b * (*gradvs)) +
                     c * (*vs) * (*vr)
                    );

 }
}
a_rs *= abs(detAe);
```

To keep the above code simple, we have assumed a constant vector field $\mathbf{b}$, and a constant source term c. The above assembler is fast, in all aspects.

The educated reader will note that, different from usual, the iterators (supporting member `begin()` and `end()` are not new classes. For instance, invoking `begin()` on tensor `grad_v_e` in $\mathbf{R}^{D\times T\times Q}$ returns a matrix in $\mathbf{R}^{D\times T}$ and not an instance of a new iterator-class containing a reference to the matrix.

Because vectors and matrices which store the computed entries must be resisable, in NumLab, at least small and large storage classes must exist. And, because NumLab uses Petrov-assemblers, also a block storage class exists.

The Second step towards an efficient implementation: For systems, ensure that component p of F, or block component `(p, q)` of its derivative can be assembled using the above fast assemblers. To this end, first note that fast assemble code above is templatised, and no longer a routine: Small vectors, matrices and tensors must be declared

```
const smallMatrix<D, T> *gradv = grad_v_e.begin();
```

using parameters of the specialisation of `Basis`. In the above example, the amount of reference basis functions `T` is used – as well as the dimension of the region of interest `D`.

The assembler code is put into two functions, a general one related to (9, and a specialised one related to 7. For the sake of demonstration we consider the assembler for the derivative, and for now focus on the case of a system of *linear differential equations*:

```
template<class Psi, class Phi>
int assembleDF(const Psi &basis_psi, const Phi &basis_phi,
               const Grid &grid,
               largeMatrix &DF, ... );
template<class Phi>
int assembleDF(const Phi &basis_phi,
               const Grid &grid,
               largeMatrix &DF, ... );
```

In each specialisation `assembleDF<Basis_1, Basis_2, Grid>(...)`, all calls to methods of `Basis` (such as `Basis::get_element_dofs()`) are **non-virtual** (see below), **and** all calls to methods of `Grid` (e.g., `get_element_vertices()`) are **non-virtual**. The latter methods are non-virtual because different `Grid` specialisations share data members of the base. `assembleDF()` just accesses these base data members. This technique is called *virtual through data member information*. For the `Grid`, `assembleDF()` must make use of such – or standard virtual methods – because `Grid` can be re-attached to `Space` during run-time.

It remains to be explained how all calls to `Basis` members can be **non-virtual**, and how `Space` delegates the assembling to `Basis`. First, observe that the user can run-time alter the collection of `Basis` (plural) (see figure 1, where `SpaceReferenceTriangleLinear` and `SpaceReference-TriangleQuadratic` are input to `Space` for the pressure and velocities). Next, observe that `assembleDF()` is parametrised with *two* `Basis`. Based on these two observations, `NumLab` uses a technique called *double dispatch*. Each module derived from `Basis`, such as `SpaceReferenceTriangleLinear`, implements a list of virtual methods (we omit the `SpaceReference-` part):

```
class TriangleLinear: public Basis
{
 ...

 int assembleDF(const Basis &basis, const Grid &g, ...)
  { return basis->assembleDF_execute(*this, g, ...) }

 int assembleDF_execute(const TriangleLinear &basis,
                        const Grid &g, ...)
  { return assembleDF<TriangleLinear>(*this, g, ...); }


 int assembleDF_execute(const TriangleQuadratic &basis,
                        const Grid &g, ...)
  { return assembleDF<TriangleQuadratic,
                      TriangleLinear>(basis, *this, g, ...); }

 ...
};
```

This shows how the finite element assemblers in `NumLab` can be fast and customisable: The `Basis` (plural) can be exchanged to different bases at run-time because the actual assembler is not present in `Space` nor in `Basis`:

```
class Space
{
 ...
```

```
int assembleDF(blockMatrix &DF, ... )
{
 for (int p = 0; p < nb_basis; p++)
 {
  for (int q = 0; q < nb_basis; q++)
  {
   DF(p, q) = basis[p]->assembleDF(basis[q], *grid,
                                   DF, ...);
  }
 }
}

...
};
```

The `NumLab` implementation is also fast because the assembler defacto runs with the routines `assemble<., .>(...)` which are specialised for the combination of `Basis` chosen at run-time. It should be noted that one `assemble<b1, b2>(...)` instance is needed for each basis `b1` $\neq$ `b2`, a small price to be paid for the ensured fast execution.

Next, the above approach is refined for system of *non-linear differential equations*, where as an added problem, in each block component `(p, q)`, *all or a few solution components of* v are required – as a rule. Because (Navier-Stokes) v consists of three components, it would seem that a two-`Basis` parametrisation of `assemble()` is not sufficient. However, a two-`Basis` parametrisation of `assemble()` still works fine, when the value of the solution components v is passed as a vector of values at the quadrature points, just as for instance the diffusion `a`, or convection `b` could be passed to `assemble()`. This turns out to the solution: At all quadrature points $\mathbf{x}_k$, `Space` ensures the pre-computation (sampling) of all values which are not related to a basis function $\phi_j^{(i)}$. For solution components – which do depend on basis functions – this task is delegated to `Basis`. This information is passed to `assembleDF()`.

For the sake of a lucid exhibition of the software framework implementation on module level, a range of details has been not discussed. The reader will note that for the non-linear case, `Space` must negotiate a cross-basis set of quadrature points, and, that in fact `assemble()` must be parametrised with this amount of points. An alternative is that different bases use the same set of quadrature points, suited for the highest degree of polynomial basis. The best software framework solution to such problems is continuously under investigation. Other not-commented on issues are the use of `Equation` inside the `assemble()`, specialised Navier-Stokes assemblers, the sampling of elements v required for multiple grid computations (in `NumLab`, also delegated by `Space` to `Basis`), etc., etc.

Readers of [1], [2], [11], [12] and other `NumLab` related papers have pointed out that the implementation of this powerful NumLab software framework would be far from trivial. This paper shows that this has been true indeed. Each small detail of the NumLab framework has been designed based on years of experience in: (1) iterative solution methods for complex non-linear systems of transient BVPs – with Finite Element, Difference, Volume discretisations, grid generation, etc; (2) visualisation of the resulting complex data-sets; (3) modern computer-language software design techniques.

The above modular software framework makes `NumLab` Finite Element computations highly customisable: The `NumLab` workbench contains a module called `SpaceReferenceTriangle`, which is a specialisation of `Basis`. This specialisation has a switch, which allows run-time switching between the different available basis specialisations. The implementation is standard: Module `SpaceReferenceTriangle` has a pointer to a specialisation of a member of `Basis`, which can be reassigned to at run-time.

Thus, summarising the facts which ensure that `NumLab` modules are both fast **and** customisable:

1. customisation: a software framework with sophisticated delegation-model;
2. discretisation mixes: dispatch techniques in combinations with templates;
3. fast execution: different storage classes for different circumstances.

Because there are just two Finite Element assembler templates (Galerkin and Petrov type) in NumLab, `NumLab` modules are not just both fast **and** customisable, but **also** simple to maintain.

5 Application design and use

This section demonstrates how the NumLab modules introduced in section 3 are combined to form a numerical simulation.

Figure 1 a shows a NumLab c++ Navier-Stokes simulation program composed with, and represented using, the NumLab graphical editor vission. The involved modules occur in quite standard groups:

1. A computational `Grid` specialisation;
2. A `Space` specialisation, involving specification of Finite Element bases `Basis`, and `BoundaryConditions`;
3. The BVP operator:
   a) A Finite Element Galerkin `Operator` specialisation, using;
   b) A Navier-Stokes `Equation`, and an optional;
   c) Predetermined solution, a specialisation of type `Function`.

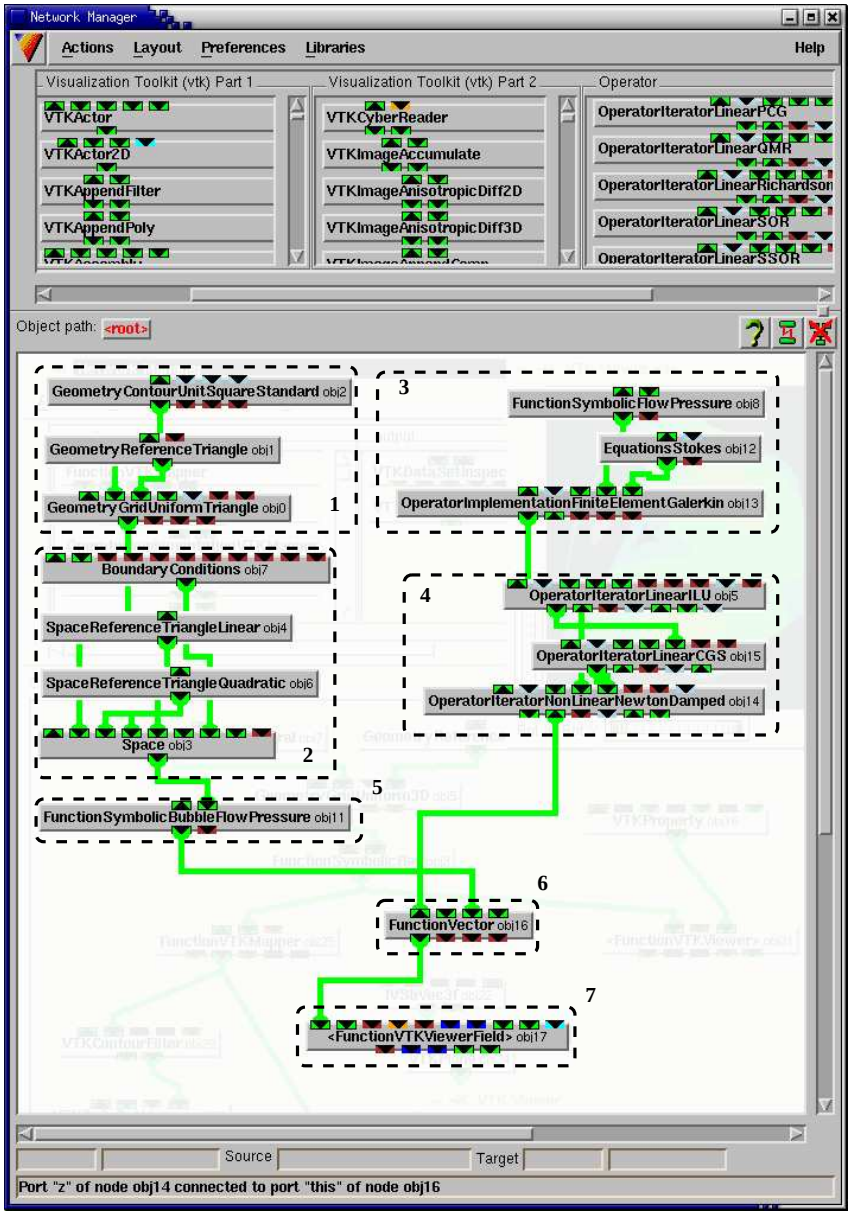


Fig. 1. A Navier-Stokes simulation built with NumLab components

4. The composed solver:
   a) A Non-linear `Solver` specialisation, using;
   b) A Linear `Solver` specialisation, using;
   c) A Preconditioner `Solver` specialisation.
5. The initial guess $v^{(0)}$ (velocities and pressure);
6. The function $v^{(k)}$ containing all subsequent iterands – velocities and pressure – (see the explanation below (1));
7. A visualisation group `FunctionVTKViewerField` for $v^{(k)}$.

Module group (2) defines the linear vector space, which contains the targeted solution (6). The operator composed in group (4) acts on this solution, i.e., it performs all transitions $v^{(k)} \mapsto v^{(k+1)}$, until all criteria are met.

The application can be extended or altered, taking modules from the libraries right at the top of figure 1, or taking modules from libraries which can be loaded during run-time. Eigenvalues of Jacobians can be visualised, etc. In this example, several NumLab libraries have been loaded, but visible are just three:

– `Visualization ToolKit 1`;
– `Visualization ToolKit 2`;
– `NumLab Operator`.

In fact, the visualisation module in figure 1 is not a single module, it is a group which contains other connected modules. Thus, when simulations get more complex, it is possible to group (hide) less important parts into one module. Non-connected input and output ports of the hidden modules become input and output ports of the new group-module. The contents of module `FunctionVTKViewerField` in figure 1 is shown if figure 2, together with the resulting output. The content of the `FunctionVTKViewerField` group is accessible after a simple double-click on the `FunctionVTKViewerField` icon. This group can be replaced with a range of available visualisation groups (stream lines, contour surfaces, contour lines, etc.).

Because NumLab adorns the original unaltered VTK and Open Inventor libraries with an interface (see section 1), its VTK and Open Inventor modules inter-mix. NumLab combines the strength of Open Inventor (superb rendering engine) with the strength of VTK (lots of high level modules, such as Image Transformations).

6 Conclusions and Future Work

As was concluded in [1], NumLab addresses two categories of limitations of current computational environments. First, NumLab builds on a few fundamental numerical notions. All entities such as iterative solver, preconditioners, time integration, Finite Element assemblers, etc., are formulated as

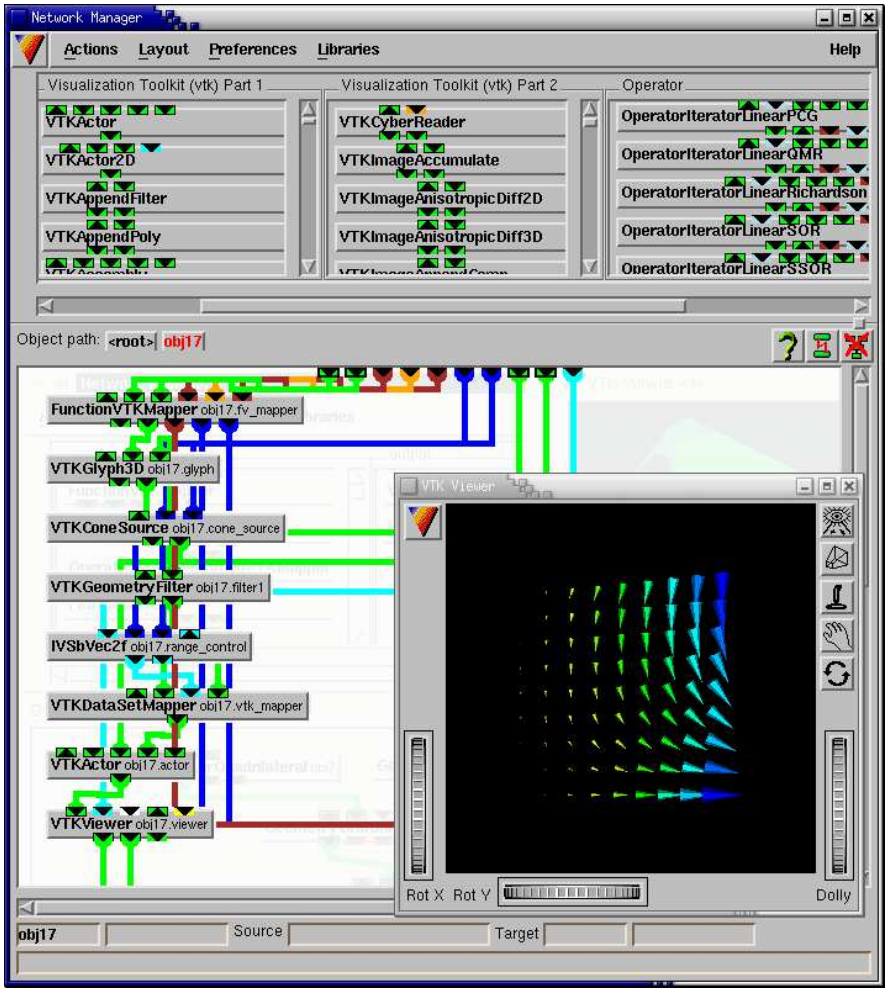


Fig. 2. The modules hidden inside group module `FunctionVTKViewerField`. The viewer at the end of the pipe-line visualises the result

operators which cause state transitions $v^{(k)} \mapsto v^{(k+1)}$. Each new operation is simple to integrate because it is of the same nature, i.e., a transition operation. Visualisation and numerical operations follow the same concepts, whence in NumLab, computations and visualisation mixes without problems.

Next, in detail demonstrated in this paper, NumLab combines a high level of *customisable features with nevertheless fast implementations*. We have shown that this is possible with a minimal amount of code, making targeted use of templates in c++. Though the amount of code is minimal –

and thus the maintenance costs are low – the larger scale required module-coworking turns out to be non-trivial pattern.

The next steps on the road towards an even more complete workbench are: merging available parallel Finite Element assemblers, integrating problem-specialised iterative solvers and preconditioners, and, started a few month ago, separating the address space of the graphical editor from the address space where all modules execute. Then failing contributed research-modules will no longer crash entire NumLab applications. The visualisation viewer module and computational modules should be run on different threads so data can be visualised while computations continue.

References

1. Maubach, J.M.L., Telea A. (accepted): The NumLab numerical laboratory for computational and visualisation. Computing and Visualisation in Science. Springer Verlag
2. Maubach J.M.L., Drenth, W.(2002): Data-Flow Oriented Visual Programming Libraries for Scientific Computing. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds) Computational Science – ICCS 2002 (LNCS 2329). Springer Verlag
3. Abram, G., Treinish, L. (1995): An Extended Data-Flow Architecture for Data Analysis and Visualisation. In: 6th Proc. IEEE Visualisation 1995, ACM Press, 263–270.
4. Anderson, E., Bai, Z., Bischof C., et al. (1995): LAPACK user's guide. SIAM, Philadelphia
5. Brun, R., Goto, M., Rademakers, F.: The CINT c/c++ interpreter `http://root.cern.ch/root/Cint.html`
6. Bruaset, A.M., Langtangen, H.P. (1996): A Comprehensive Set of Tools for Solving Partial Differential Equations: Diffpack. In: Daehlen, M., Tveito, A. (eds) Numerical Methods and Software Tools in Industrial Mathematics. Springer Verlag
7. Gunn, C., Ortmann, A., Pinkall, U., Polthier, K., Schwarz, U. (1996): Oorange: A Virtual Laboratory for Experimental Mathematics, Sonderforschungsbereich 288, Technical University Berlin. `http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html`
8. Jackie, N., Davis, T., Woo, M. (1993): OpenGL Programming Guide. Addison-Wesley
9. Schroeder, W., Martin, K., Lorensen, B. (1995): The Visualisation Toolkit: An Object-Oriented Approach to 3D Graphics. Prentice Hall
10. Stroustrup, B. (1997): The c++ Programming Manual (3rd edition). Addison-Wesley
11. Telea A., van Wijk, J.J. (1999): VISSION: An Object Oriented Dataflow System for Simulation and Visualisation. In: Grller, E., Lffelmann, H., Ribarsky, W. (eds) Proceedings of IEEE VisSym 1999. Springer Verlag
12. Telea A. (1999): Combining Object Orientation and Dataflow Modeling in the VISSION Simulation System. In: Proceedings of TOOLS'99 Europe. IEEE Computer Society Press

13. Upson, C., Faulhaber, T., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., van Dam, A. (1989): The Application Visualisation System: A Computational Environment for Scientific Visualisation. IEEE Computer Graphics and Applications, 30–42
14. Wernecke, J. (1993): The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor. Addison-Wesley
15. The Open Source Software Project Based on IBM's Visualisation Data Explorer. `http://www.opendx.org/`
16. IMSL (1987): FORTRAN Subroutines for Mathematical Applications, User's Manual. IMSL
17. SEPRA Analysis `http://ta.twi.tudelft.nl/sepran/sepran.html`
18. Matlab (1992): Matlab Reference Guide. The Math Works Inc.
19. Wolfram., S. (1999): The Mathematica Book 4-th edition. Cambridge University Press
20. NAG (1990): FORTRAN Library, Introductory Guide, Mark 14. Numerical Analysis Group Limited and Inc.
21. INRIA-Rocquencourt (2000): Scilab Documentation for release 2.4.1. `http://www-rocq.inria.fr/scilab/doc.html`